Peter Desain and Henkjan Honing
Center for Knowledge Technology
Utrecht School of the Arts
Lange Viestraat 2B
NL-3511 BK Utrecht, The Netherlands
and
COCO Foundation
P.O. Box 1037
NL-3500 BA Utrecht, The Netherlands

Time Functions Function Best as Functions of Multiple Times

This article presents an elegant way of representing control functions at an abstract level. It introduces time functions that have multiple times as arguments. In this way the generalized concept of a time function can support absolute and relative kinds of time behavior. Furthermore the possibilities of composition and transformation of time functions themselves is retained. The proposed solution has three main advantages. First, for the human user the language is transparent, and no unforeseen interactions or side effects take place. Second, it is independent of host language and composition system and can be used in a variety of known environments (even in real-time systems). Finally, the method is easy to adapt to run on parallel architectures, where each note can be handled by a different processor without the need for information passing between them.

Background

In the early history of computer music composition (Loy 1988), the systems available either took a monolithically continuous, signal-processing-inspired approach (Mathews and Moore 1970; Berg 1979), or used a discrete, note- or event-based technique (Hiller, Leal, and Baker 1966; Koenig 1970). Although some early work stressed the importance of hybrid systems (Mathews 1969; Buxton et al. 1978), this division became even more obvious once the rich domain of hardware and software that became available for MIDI lured designers into building composition systems close to this note-based protocol.

MIDI allows for some rudimentary continuous control (e.g., polyphonic aftertouch), but most parameter changes affect either all sounding notes or all notes on a specific channel (Loy 1985). With the advent of inexpensive signal processing hardware that allows more natural continuous control over parameters during each note's evolution, the quest for elegant constructs in composition languages supporting both worlds is on again. Some researchers foresaw these developments and made attempts to bridge the gap between these two worlds by stating the problems (Dannenberg et al. 1989; Huron 1990; Honing 1991), and proposing solutions to them (Dannenberg, McAvinney, and Rubine 1986; Anderson and Kuivila 1989).

The main problem that arises is how continuous control functions should behave under specification and transformation of the discrete structure. A notorious example is the vibrato problem—the vibrato on a note should not slow down if the note itself is elongated, rather some extra vibrato cycles should be added to the pitch envelope. A discrete analogue of the vibrato problem is the drumroll, which should extend when its duration is prolonged, with more drum beats added, but whose rate should not slow down. A glissando, however, specified by the same means of a continuous pitch envelope, should be stretched along with the note duration. An ornament (e.g., a mordent) is invariant under transformation of the duration of the note.

Several solutions for these problems have been stated, but all are unsatisfactory. We will describe a few of the proposals below. In Canon (Dannenberg

Authors' current addresses: Peter Desain, NICI Nijmegen University, P. O. Box 9104, NL-6500 NE Nijmegen, The Netherlands. Henkjan Honing, University of Amsterdam, Computational Linguistics, Spuistraat 134 NL-1012 VB, Amsterdam, The Netherlands

Computer Music Journal, Vol. 16, No. 2, Summer 1992,

1989) a collection of transformations on a fixed set of attributes is used, together with a way of communicating environment information to new transformations.With these constructs he explicitly solves the drum roll problem, though in a nontrivial, almost procedural way. Dannenberg proposes the term *behavioral abstraction* for the ability to express these complex parametrized behaviors.

Anderson and Kuivila (1989) propose a solution based only on global time, prohibiting the composer from thinking in terms of local constructs; most real-time composition systems have, besides actual time, no sense of time at all (see Desain and Honing in preparation). Solutions proposed for object-oriented composition systems (e.g., Pope 1989, 1991) suffer from a declarative/procedural confusion whereby transformations and musical objects form no orthogonal sets; each new transformation added has to take all object types and all existing transformations into account. This inevitably leads to the situation whereby some transformations cannot be done twice or some combinations cannot be done in an arbitrary order.

In this article we present an elegant and—once understood—obvious way of representing control functions at a more abstract level that simply avoids all these problems and yields a transparent specification of (discrete) musical structures with continuous control over their parameters. To be able to illustrate this approach, a simple framework language for discrete musical objects and their ordering in time is given first, expressed in Common Lisp (Steele 1990).

A Framework for Discrete Musical Objects

Let us start by assuming a basic *note* object, and a basic *rest object* (called *pause,* to avoid clashing with the Common Lisp primitive **rest**), with some parameters specified by keywords:

```
(note :duration 2 :pitch 60 :amplitude 0.8)
(pause :duration 1)
```

The syntax is taken directly from Lisp, that is, prefix notation with the function name before the arguments, the whole expression being enclosed in parentheses. The arguments are specified as pairs of keywords and values. The actual parameters allowed are irrelevant for the present introduction—a MIDI-based composition system might need different parameters than a signal processing approach. Furthermore, the discussion of the magnitude scales for these parameters is ignored here, a simple assumption of a duration scale in seconds, a pitch scale in MIDI key numbers (with fractional part), and a [0,1] scale for amplitude is assumed in the examples. Even the semantics of such expressions—whether they initiate processes, deliver data structures that represent musical objects (i.e., event-lists), or are the actual procedures that output the material directly—is immaterial here. One possible implementation of this notation is given in the Appendix.

We assume that unset parameters are defaulted to reasonable values (duration 1 second, pitch 60 [middle-c], and amplitude 0.7), for ease of use in the examples. It must be possible to specify the timing of the basic musical objects, either by passing parameters for start time and duration directly (as is used here for the duration parameter), by means of parameters that place or move objects in time (Abbott 1981; Dannenberg, McAvinney, and Rubine 1986; Balaban 1989), or by *constructor* functions that build or play musical objects in a distinct time order (Smoliar 1980; Dannenberg 1989). For the sake of simplicity, we will use the last approach with the *Sequential* and *Parallel* constructs that we introduced in the LOCO composition language (Desain and Honing 1988), which were subsequently elaborated as a basis for transformations (Desain 1990). These constructs mirror the sequential and parallel execution primitives used in parallel language design. *S* stands for a sequential ordering of its components, the whole structure ending after the last one, and *P* stands for a parallel structuring ending at the end time of its longest component. As an example, consider the musical object declared in Fig. 1a. Its graphical piano-roll-like representation is shown in Fig. 1b, where time is represented on the *x* axis, pitch is represented on the *y* axis, and the amplitude of a note is represented by its shading.

An extension of this approach allows for high-level timing control for defining nonstandard musical objects such as grace notes. As these constructs

Fig. 1. Example of a time structured musical object (a) and graphical notation of the same musical object (b).

Fig. 2. Use of procedural abstraction in defining a compound musical object (a) and graphical notation of the same musical object (b).

```
(s (p (note :duration 2 :pitch 62 :amplitude 1.0)
      (note :duration 4 :pitch 65 :amplitude 0.7))
   (note :duration 5 :pitch 58 :amplitude 0.3)))
```

Fig. 1a

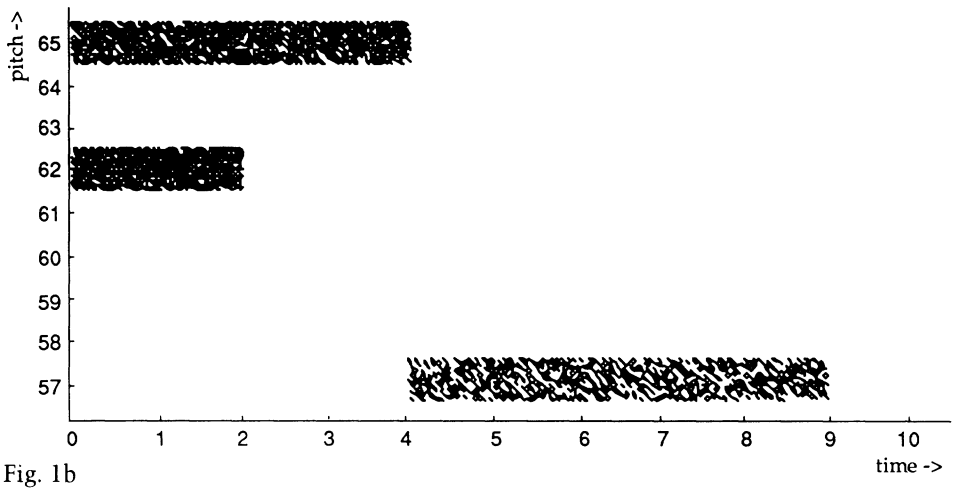

Fig. 1b

```
(defun major-chord (duration key)
  (p (note :duration duration :pitch key)
     (note :duration duration :pitch (+ key 4)))
     (note :duration duration :pitch (+ key 7))))

(s (major-chord 2 57) (major-chord 5 58) (major-chord 2 57))
```

Fig. 2a

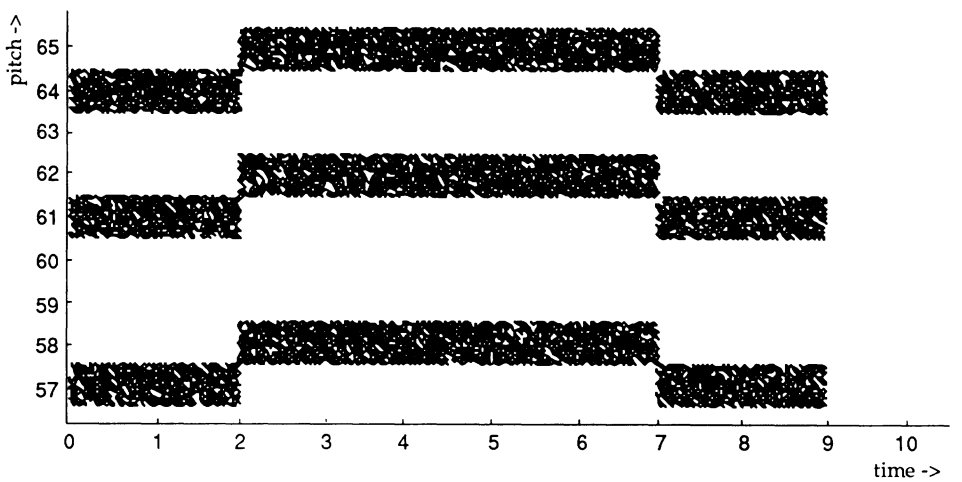

Fig. 2b

are not essential for the present argument, we refer to Desain and Honing (1988) for more details. For naming complex musical objects, forming parameterized families of objects, and defining musical transformations, we will use the standard procedural abstraction (function definition) facilities of the host language (Lisp). An example of a compound musical object built by those means is shown in Lisp source and graphically in Fig. 2.

Next we introduce a basic transformation on the timing of fully constructed musical objects that we can use to demonstrate the behavior of the time functions when the duration of the object to which they are linked is changed. The **stretch** transformation multiplies each duration of the enclosed objects by a given factor.

```
(stretch (note :duration 1 :pitch 60 :amplitude 1) 2)
 → (note :duration 2 :pitch 60 :amplitude 1)
```

These preliminary constructs constitute a world rich enough to introduce continuous time functions and their problems, but the same solution can be used in most present composition systems, however different their notion of musical objects and collections thereof, and whatever way the time relations between them are specified.

Continuous Control

The Problem

A natural thought, when bored with note-based discrete systems, is to pass each note a continuously variable function of time as parameter, for example, for pitch or amplitude, instead of a constant value. The functions passed are functions of the actual time, and elegant ways to build and transform them can be given. Often, though, these functions have been regarded as control signals (resembling audio signals) even up to the point where a list of data points, interpreted at a fixed (low) sample rate has been called a "function" (e.g., Schottstaedt 1983; Puckette 1988), obscuring the highly abstract powerful possibilities of functions in their mathematical sense (exceptions are Rodet and Cointe 1984; Dannenberg, McAvinney, and Rubine 1986. Anderson and Kuivila [1989] propose an alternative). But even when the full power of function specification is used, time functions are considered to be functions of actual time. This complicates the coupling of these functions to discrete objects and gave rise to the problem described above—the notorious vibrato problem and its discrete counterpart, the drumroll problem.

A related concern is the use of a relative or absolute start point for the time base used. Composers sometimes prefer the use of an absolute time scale because of the (false) impression of total control. However, it implies that an envelope be redefined each time it is used at another absolute point in time. This can be avoided simply by using a time base relative to the object under construction. This, of course, does not mean that the notion of absolute time control can be ignored. It is indispensable when time relations with events outside the musical piece (e.g., midnight church bells) are to be taken into account, or, as is the case more often, when relations between different and independent musical objects have to be maintained (e.g., synchronized vibrato between different voices).

A Solution

The solution we propose is to make each control function a function of more parameters, each parameter reflecting a different aspect of time. As an example, we will develop this notion for time functions of three parameters, the absolute start time of the discrete musical object it is controlling, the absolute time duration of this musical object, and a relative progress, expressed as how much time has elapsed since the start time, relative to the duration (a number in the range [0,1]). Other choices are, of course, possible here (e.g., start time, end time, and actual time). All these parameters will be passed their appropriate values automatically by the interpretative system. With this definition of a time function, the user can now choose to use some time parameters and ignore others to make time functions that behave differently when used for musical objects with different duration or start time.

As a first example, let us define an elastic ramp control, independent of an absolute start time, taking the full duration of the musical object to reach

Fig. 3. Definition of a ramp time function and a musical object using it (a) and glissandi that extend when stretched in time given (b).

```
(defun ramp (from to)
  #'(lambda (start duration progress)
      (+ from (* progress (- to from)))))

(defun glissando-example ()
  (s (note :duration 1 :pitch 58)
     (p (note :duration 2.5 :pitch (ramp 64 61))
        (s (pause :duration .5)
           (note :duration 2 :pitch (ramp 61 60))))))

(s (glissando-example) (stretch (glissando-example) 2))
```

Fig. 3a

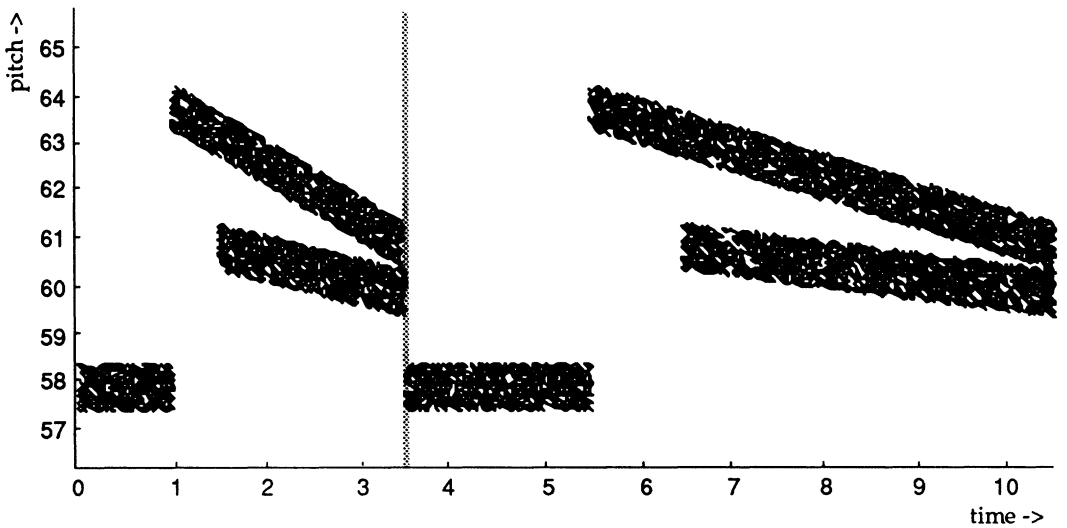

Fig. 3b

its final value. We will use it to control the pitch of some notes, creating glissandi. The function **ramp** produces a linear time function of the three time parameters mentioned above. It ignores the absolute start time and duration parameters, depending only on the progress of the evolving note (a number between 0 and 1) to calculate its value. Figure 3 shows how the same **ramp** construct is used for notes of different duration—yielding an appropriate glissando—and how the musical idea is kept intact under a **stretch** transformation.

Now let us construct, with the same means, a vibrato function, parameterized by the fundamental pitch it is to be applied to, and the frequency and depth of the vibrato itself. It only depends on the actual time elapsed during the musical object—the product of duration and progress. Now the application of the vibrato function to notes of different duration will not alter the vibrato rate, nor will the **stretch** transformation applied to the compound musical object (see Fig. 4b).

If we make a similar sinusoidal glissando function, expressed in terms of relative time (progress), a **stretch** of the musical object will slow down the rate (see Fig. 4c).

Absolute time can also be used to make time functions that are not influenced at all when they are applied to musical objects with a different dura-

Fig. 4. Definition of musical objects using vibrati, glissandi, and ornaments (a); vibrati that elongate when stretched in time (b); sinusoidal glissandi that extend when stretched in time (c); and sinusoidal ornament not affected when stretched in time (d).

```
(defun sine-oscillator (offset frequency depth)
  #'(lambda (start duration progress)
      (+ offset
         (* depth
            (sin (* 2 pi duration progress frequency))))))

(defun sine-glissando (key depth)
  #'(lambda (start duration progress)
      (+ key
         (* depth (sin (* 2 pi progress))))))

(defun sine-ornament (key count)
  #'(lambda (start duration progress
                  &aux (relative-time (* duration progress)))
      (+ key
         (if (< relative-time count)
           (sin (* 2 pi relative-time))
           0))))

(defun vibrato-example ()
  (s (note :duration 1 :pitch 58)
     (p (note :duration 2.5 :pitch (sine-oscillator 64 1 .5))
        (s (pause :duration .5)
           (note :duration 2 :pitch (sine-oscillator 61 1 1))))))

(defun glissando-example ()
  (s (note :duration 1 :pitch 58)
     (p (note :duration 2.5 :pitch (sine-glissando 64 .5))
        (s (pause :duration .5)
           (note :duration 2 :pitch (sine-glissando 61 1))))))

(defun ornament-example ()
  (s (note :duration 1 :pitch 58)
     (p (note :duration 2.5 :pitch (sine-ornament 64 .5))
        (s (pause :duration .5)
           (note :duration 2 :pitch (sine-ornament 61 1))))))

(s (vibrato-example) (stretch (vibrato-example) 2))      ; Figure 4b
(s (glissando-example) (stretch (glissando-example) 2)) ; Figure 4c
(s (ornament-example) (stretch (ornament-example) 2))   ; Figure 4d
```

Fig. 4a

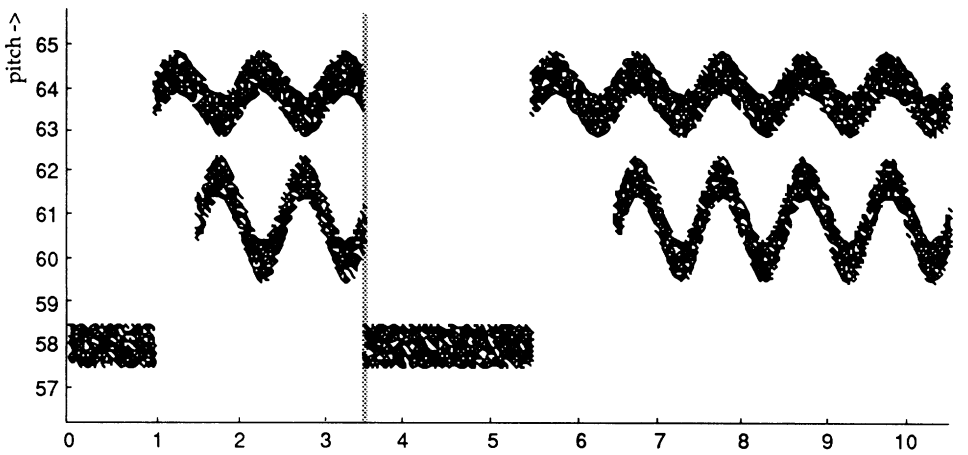

Fig. 4b

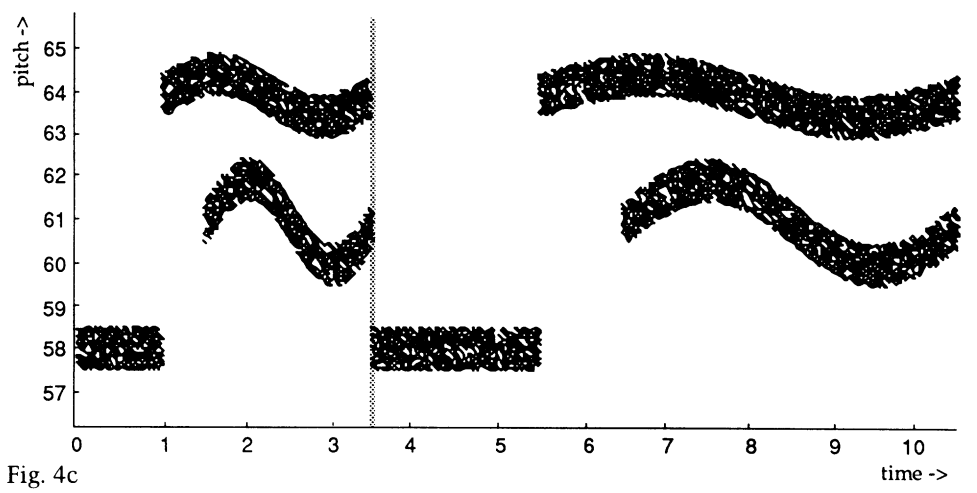

Fig. 4c

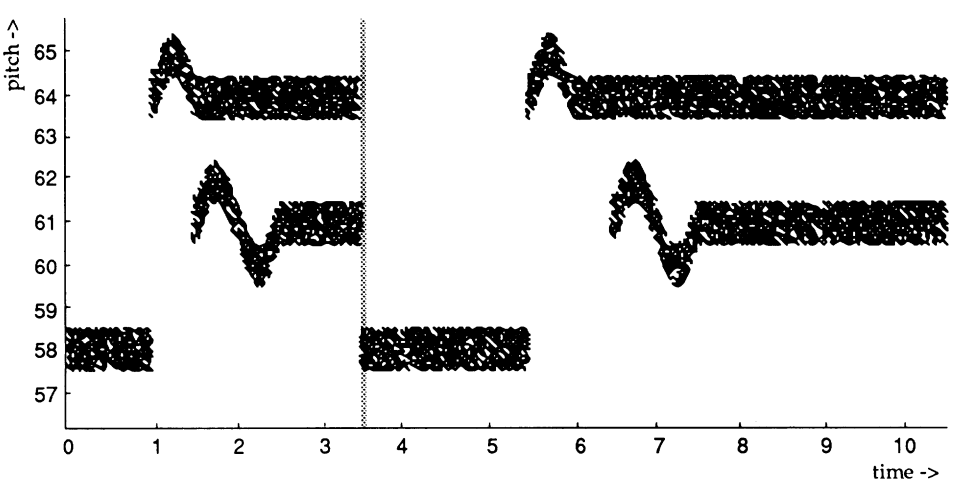

Fig. 4d

Fig. 5. Surfaces representing vibrato (a), glissando (b), and ornament (c) time functions as a function of duration and relative time.

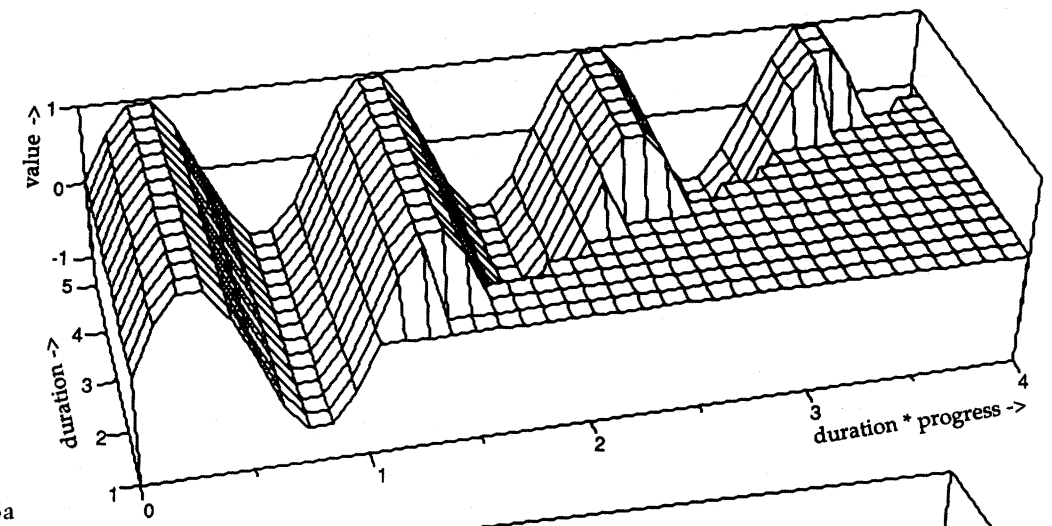

Fig. 5a

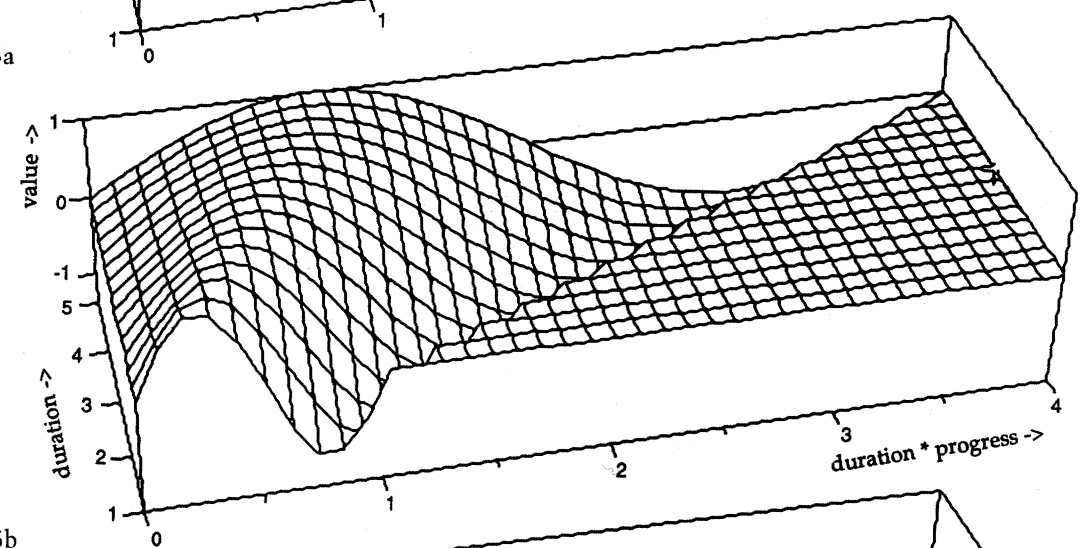

Fig. 5b

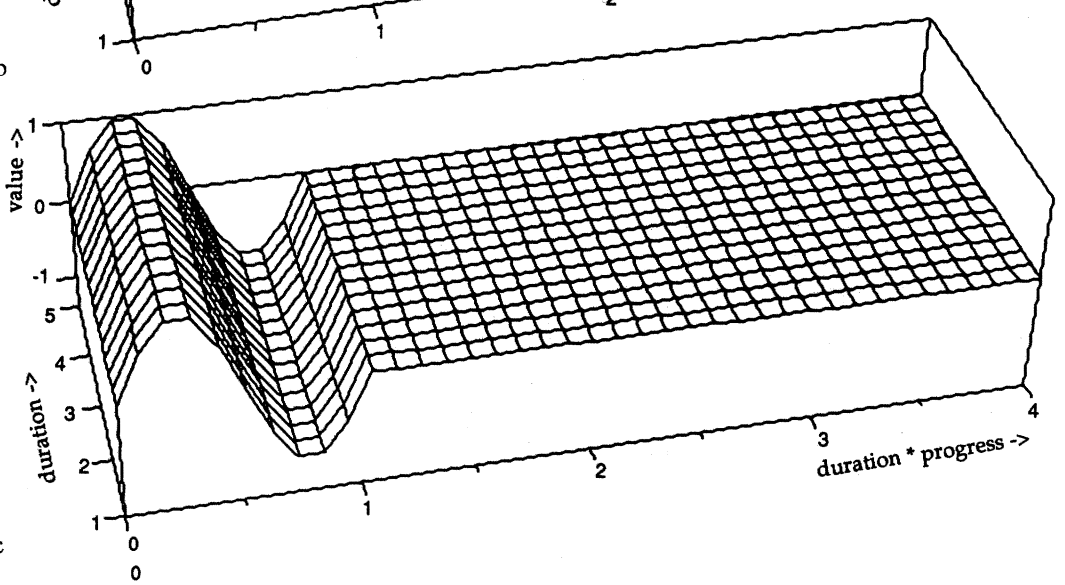

Fig. 5c

Fig. 6. Example of a musical object using a synchronized vibrato (a) and graphical notation of the same musical object (b).

```
(defun sync-oscillator (offset frequency depth &optional (phase 0))
  #'(lambda (start duration progress)
      (+ offset
         (* depth
            (sin (* 2 pi (+ phase
                            (* start frequency)
                            (* duration progress frequency))))))))

(defun synchronized-vibrato-example ()
  (s (note :duration 1 :pitch 58)
     (p (note :duration 2.5 :pitch (sync-oscillator 64 1 .5))
        (s (pause :duration .5)
           (note :duration 2 :pitch (sync-oscillator 61 1 1))))))

(s (synchronized-vibrato-example)
   (stretch (synchronized-vibrato-example) 2))
```

Fig. 6a

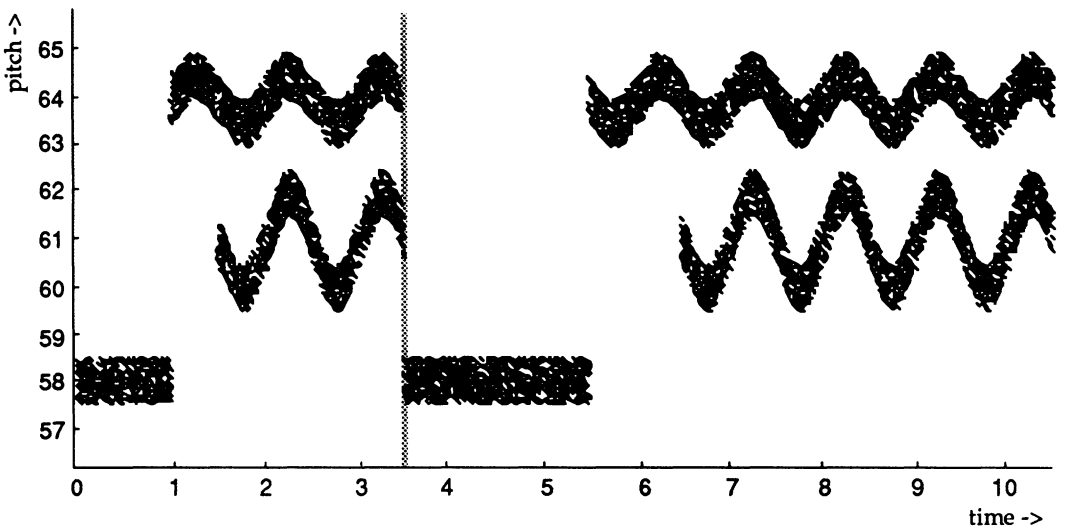

Fig. 6b

tion. An ornament could have such character; a sinusoidal ornament will keep its absolute timing when stretched (see Fig. 4d).

The abstract vibrato, glissando, and ornament time functions can be depicted nicely in a three-dimensional surface plot, as is shown in Fig. 5. The relative time (duration * progress) and the duration are used as dependent variables here. For any note duration, the actual time function used will be a cross-section of these surfaces.

Using the absolute start-time parameter enables full control over timing with respect to a global clock. This can be used to specify the phase of a vibrato among parallel notes, such that they can be synchronized, as is shown in Fig. 6 (compare with Fig. 4b).

This completes the examples of the use of generalized time functions with multiple parameters as control functions for individual basic objects. It shows how problems of synchronization and time

Fig. 7. Example of applying the same local envelope function to the individual note objects (a) and graphical notation of the musical object (b).

```
(defun envelope-example ()
  (let ((envelope (ramp 0 1)))
    (s (note :duration 1 :pitch 58 :amplitude envelope)
       (p (note :duration 2.5 :pitch 64 :amplitude envelope)
          (s (pause :duration .5)
             (note :duration 2 :pitch 61 :amplitude envelope))))))

(s (envelope-example) (stretch (envelope-example) 2))
```

Fig. 7a

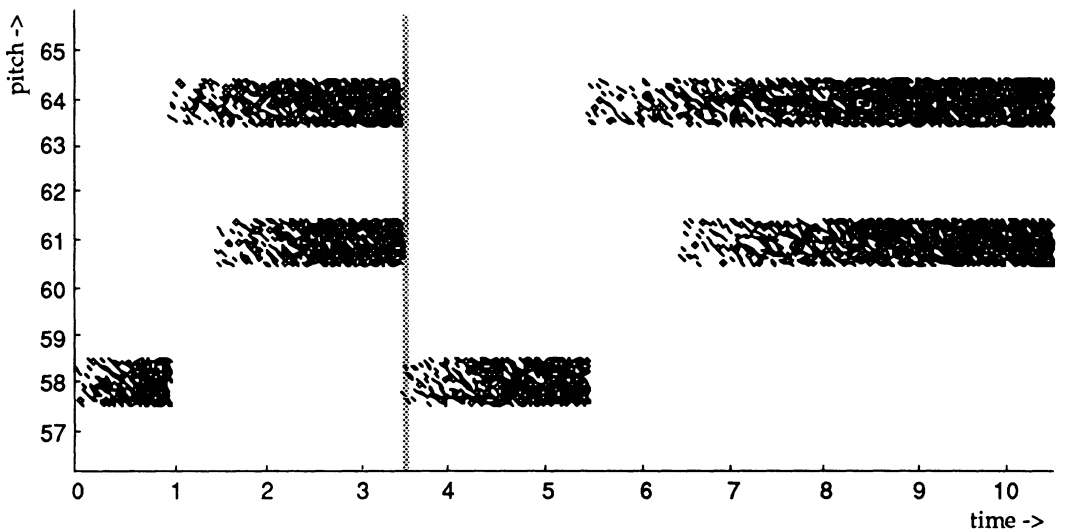

Fig. 7b

modification are elegantly avoided by lifting the concept of a time function to a more abstract level. Of course the control functions used here are rudimentary in their musical value—much more elaborate envelopes are needed—but they can all be based on the same idea, and we show below how simple time functions can be combined into complex ones, but first we want to tackle the problem of time functions extending over a collection of several musical objects.

Control Over Compound Objects

In composition the use of time-dependent control specified over a collection of musical objects is abundant. The simplest example specifies the same control information to be applied to each basic object. Naming a control function (as done with the **let** local binding construct of Common Lisp) is then natural (see Fig. 7).

A different method has to be used to pass time functions evolving over a compound musical object, to each basic musical object. An example of such a construct is a crescendo from a certain loudness level to another, starting at the start of the musical structure it is applied to, and extending over its total duration. We need to introduce a new construct in the language to enable the passing of information from collections of musical objects to such envelopes. It follows the same syntax as the **let** construct but modifies the time functions bound with it such that they will behave appropriately. In

Fig. 8. Example of applying a global crescendo function to the individual note objects (a) and graphical notation of the musical object (b).

```
(defun crescendo-example ()
  (let-time-fun-over-compound ((crescendo (ramp 0 1)))
    (s (note :duration 1 :pitch 58 :amplitude crescendo)
       (p (note :duration 2.5 :pitch 64 :amplitude crescendo)
          (s (pause :duration .5)
             (note :duration 2 :pitch 61 :amplitude crescendo))))))

(s (crescendo-example) (stretch (crescendo-example) 2))
```

Fig. 8a

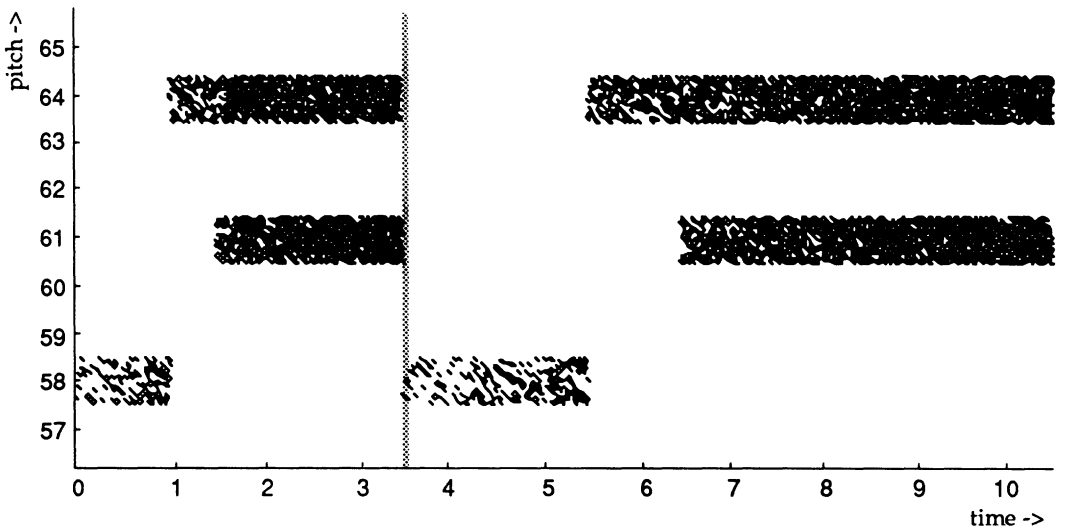

Fig. 8b

Fig. 8 the definition of a crescendo is shown using the same **ramp** function and the same musical structure as used in Fig. 7.

Time Function Composition

Building a comprehensive set of musically useful time functions can best be done by supplying some simple, basic time functions and some ways of building complex ones by transforming and combining them. The function **time-fun-compose** is one of the higher-order functions that can be imagined that supports this. It generalizes any operation to the corresponding combination of the results of time functions. The example in Fig. 9 shows both an additive combination of an oscillator and a ramp time function for pitch, and one using a multiplicative combination for the amplitude parameter. An object-oriented approach here, packaging time functions in their own class and overloading the stan-

Fig. 9. Example of combining time functions (a) and graphical notation of the musical object (b).

```
(defun compose-example ()
  (note :duration 7
        :pitch (time-fun-compose #'+ (oscillator 58 1 1) (ramp 5 0))
        :amplitude (time-fun-compose #'*
                                     (oscillator .5 1 .5)
                                     (ramp 1 0))))

(s (compose-example) (stretch (compose-example) .5))
```

Fig. 9a

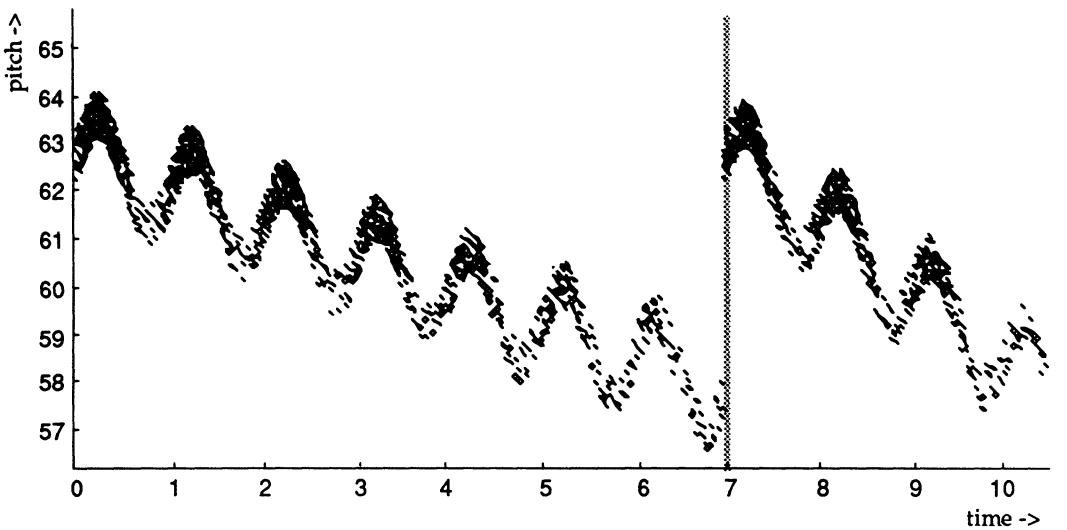

Fig. 9b

dard arithmetic operations for them, would, of course, simplify the syntax.

Another useful combination is the concatenation of two time functions, with an extra argument expressing the proportion of the duration handled by the first, implying the remaining time for the second. In Fig. 10 the concatenation of two ramp envelopes is shown, one used locally for the pitch, the other used globally for the amplitude parameters.

An ever richer world of possibilities opens up when time functions accept time functions as arguments; their parameters may then change over time as well. In the example we use a sine oscillator that changes its frequency over time, controlled by a ramp time function. A new definition of an oscillator that takes functional arguments can easily be constructed. In Fig. 11a such a sine oscillator time function is defined (it uses the function **time-funcall,** and the function **make-sine** that supplies a sine function that remembers its state over time).

It has to be stressed here again that these combinations of time functions preserve—in a compound way—the different ways in which their constituent components deal with time. For instance, the composition of a glissando and a vibrato, as in Fig. 9b, can still be stretched in time consistently; the vibrato will add cycles at the same rate and the glissando will slow down. This composibility of behavior is an important characteristic of this solution.

Fig. 10. Example of concatenating time functions (a) and concatenations of a crescendo and a decrescendo, and upward and downward glissandi (b)

```
(defun crescendo-decrescendo-example ()
  (let-time-funs-over-compound
   ((cresc-decresc (time-fun-concatenate (ramp 0 1)(ramp 1 0) .2)))
   (let ((glissando
          (time-fun-concatenate (ramp 58 59)(ramp 59 58) .8)))
     (s (note :duration 1 :pitch glissando :amplitude cresc-decresc)
        (p (note :duration 2.5 :pitch 64 :amplitude cresc-decresc)
           (s (pause :duration .5)
              (note :duration 2
                    :pitch 61
                    :amplitude cresc-decresc)))))))

(s (crescendo-decrescendo-example)
   (stretch (crescendo-decrescendo-example) 2))
```

Fig. 10a

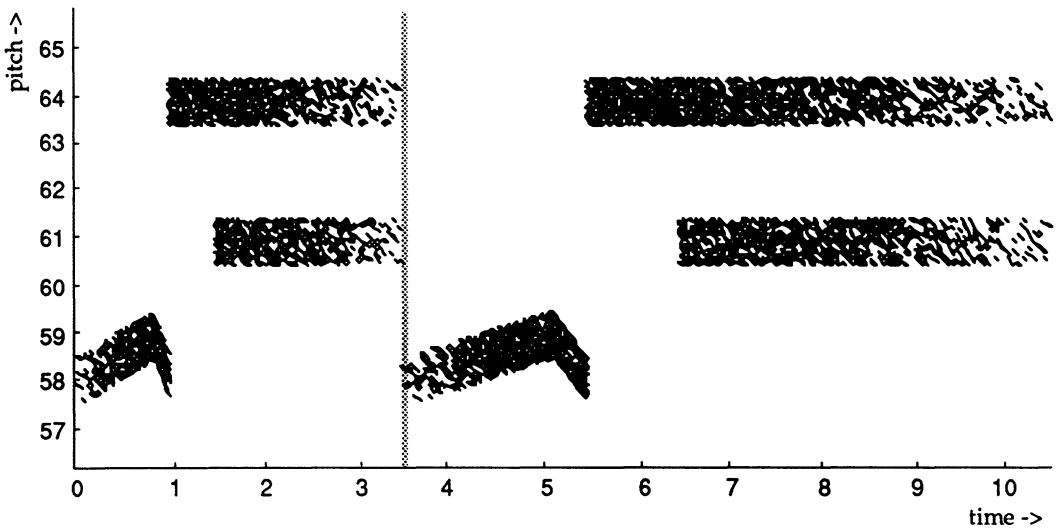

Fig. 10b

Specification by Means of Transformation

The reader might now wonder how transformations of a complex musical object can be taken care of. Because of the abstractions chosen in this design, this almost comes for free (in contradiction to other systems; see the closing remarks in [Dannenberg 1989] and [Rahn 1990]). Transformations are just another way to specify complex musical objects.

Below a set of transformations and their equivalents are shown.

```
(stretch
   (amplitude
     (s (note) (transpose (note) 2))
     (ramp 1 0))
 2)
 →
(let-time-funs-over-compound ((envelope (ramp 1 0)))
  (s (note :duration 2 pitch 60 :amplitude envelope)
     (note :duration 2 pitch 62 :amplitude envelope)))
```

Fig. 11. Example using an oscillator taking functional arguments (a) and graphical notation of the musical object (b).

```
(defun sine-osc (frequency)
  (let ((sine (make-sine)))
    #'(lambda (start duration progress
                    &aux (time (+ start (* duration progress))))
        (funcall sine time
                 (time-funcall frequency start duration progress)))))

(defun new-oscillator (offset frequency depth)
  (time-fun-compose #'+ offset
                    (time-fun-compose #'* depth
                                      (sine-osc frequency))))

(defun new-vibrato-example ()
  (s (note :duration 1 :pitch 58)
     (p (note :duration 2.5
              :pitch (new-oscillator 64 2 (ramp 0 1)))
        (s (pause :duration .5)
           (note :duration 2
                 :pitch (new-oscillator 61 (ramp 0 1) 1)
                 :amplitude (new-oscillator .5 (ramp 0 1) .5))))))

(s (new-vibrato-example) (stretch (new-vibrato-example) 2))
```

Fig. 11a

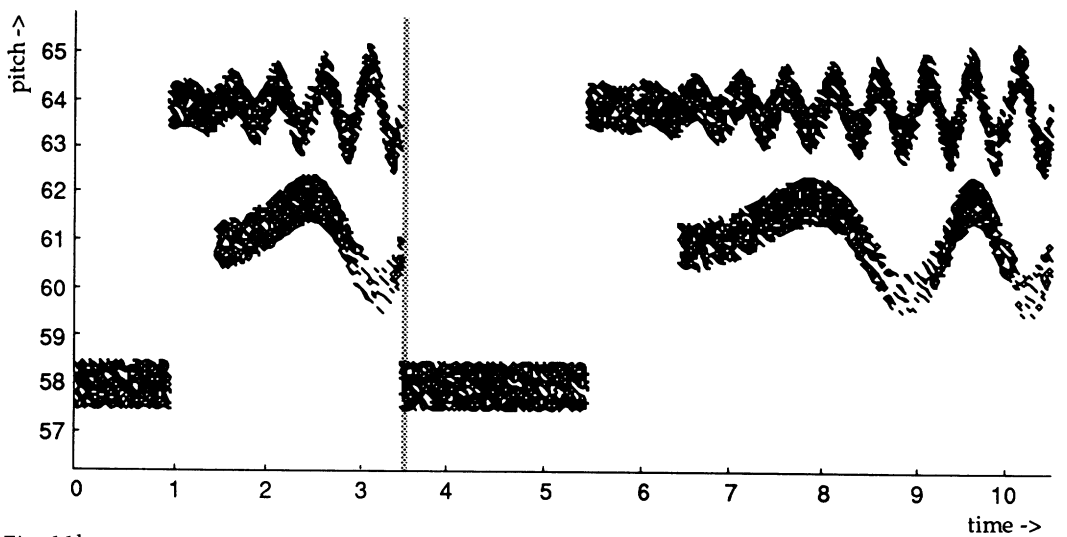

Fig. 11b

Fig. 12. Examples of attribute transformations (a) and graphical notation of the musical object (b).

```
(defun transpose (object interval)
  (attribute-transform :pitch interval #'+ object))

(defun amplitude (object amplitude)
  (attribute-transform :amplitude amplitude #'* object))

(s (transpose (new-vibrato-example) -1)
   (amplitude (stretch (new-vibrato-example) 2)
              (ramp 1 0)))
```

Fig. 12a

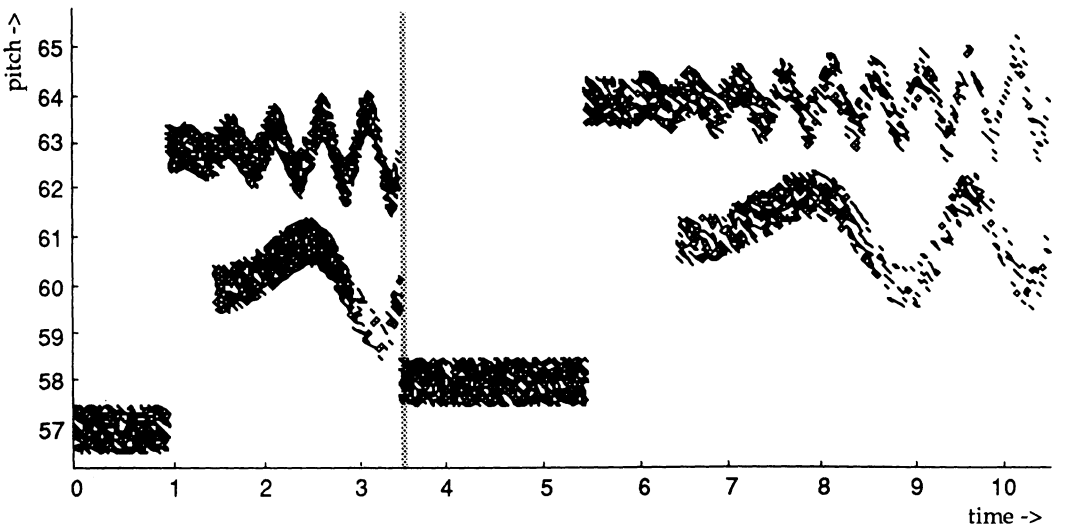

Fig. 12b

A free mixture of direct specification and transformation can, of course, be used. In Fig. 12 two examples of simple transformations are given. The **attribute-transform** function supports these kinds of transformations on the attributes of the notes. It takes, besides the musical object applied to, a keyword identifying an attribute, a time function or constant, and an operator used in combining the results of the time function with the time functions or constants found in the musical object.

Microworld

To enable the reader to test and experiment with these ideas, a simple implementation of a language with these capabilities is given the Appendix; the full implementation is part of our COCO composition system. The basic and compound musical objects are defined procedurally. Because their context (i.e., their start time) is still unknown at the time of their declaration, they will deliver functions (called event-list generators) of this context that will, when given a start time and a scale factor, return an event-list together with its end time.

A **draw** procedure is available to transform such musical objects into the graphical representation that was used in drawing the figures. Low-level **draw** routines for the user's window system or plotter, implementing the **draw-air-brush** primitive still have to be supplied. The incorporation of a **play** primitive to sort event-lists and send data to a MIDI driver is left to the reader as an exercise. Note that

this is a nontrivial task since some technical tricks (e.g., allocating notes to different MIDI channels) must be used to allow continuous control of the individual note parameters.

Extensions

Articulation

Some parameters are not continuously variable by nature (e.g the articulation of a note—the proportion of its duration that it is actually sounding), but they can be modeled well by using continuously variable time functions extending over compound musical objects, sampled once automatically by the system at the objects' start time (as in [Dannenberg 1989]). It is even possible to supply this information as an extra argument to all time functions (at the risk of becoming circular, articulation time functions themselves should not be allowed to use the articulation parameter). This elegantly solves the specification of, for example, different attack, decay, and release sections of envelope functions in terms of an articulation factor.

Real-Time Control

There is no reason why this approach could not be used in real-time control. If it is possible in the host system to specify the start of a musical object (e.g., a *note-on* command) without its end, then the set of parameters passed to time functions has to be adapted (e.g., to absolute start time, and absolute time elapsed after the start). Time functions can naturally depend on incoming (real-time) parameters if their evaluation is postponed until the last possible moment. Note that in that case time functions are not strictly functional any more, reading control signals from input ports.

Rubato Functions and Time Maps

It might be possible to add flexibility to the common rubato functions and nested time maps used for composition systems (Jaffe 1985; Desain and Honing 1992) by supplying them with multiple time arguments as well. To avoid circularity a division in score and performance time has to be made. This could then be a basis for meaningful specifications and transformations of expressive timing. The possibilities have yet to be investigated.

Advantages of "Generalized Time Functions"

Generalized time functions have three main advantages. First, for the human user the language is transparent, and no unforeseen interactions or side effects take place. Second, musical objects, time functions, transformations of musical objects are orthogonal sets; they serve as a solid and extensible basis for further design, independent of host language or composition system. Finally, from the hardware perspective this approach has the distinct advantage of being easy to adapt to run on parallel architectures; each note can be handled by a different processor, without the need for information passing between them. Note-based parallelism seems the most promising distribution of labor for most parallel architectures.

Acknowledgments

Thanks to Stephen Pope for the enlightening discussions on music representation during his stay at the Center for Knowledge Technology in 1990, and to Shaun Stevens for his corrections to an earlier version of this text. Roger Dannenberg deserves special thanks for his very useful comments.

References

Abbott, C. 1981. "The 4CED Program." *Computer Music Journal* 5(1): Reprinted in C. Roads, ed. 1989. *The Music Machine.* Cambridge: MIT Press, pp. 311–332.

Anderson, D. A., and R. Kuivila. 1989. "Continuous Abstractions for Discrete Event Languages." *Computer Music Journal* 13(3): 11–23.

Balaban, M. 1989. "Music Structures: A Temporal-Hierarchical Representation for Music." *Musikometrika* 2:

Berg, P. 1979. "Pile: A Language for Sound Synthesis." *Computer Music Journal* 3(1): Reprinted in C. Roads and J. Strawn, eds. 1985. *Foundations of Computer Music.* Cambridge: MIT Press, pp. 160–187.

Buxton, W., et al. 1978. "The Use of Hierarchy and Instance in a Data Structure for Computer Music." *Computer Music Journal* 2(4): Reprinted in C. Roads and J. Strawn, eds. 1985. *Foundations of Computer Music.* Cambridge: MIT Press, pp. 443–466.

Dannenberg, R. 1989. "The CANON Score Language." *Computer Music Journal* 13(1): 47–56.

Dannenberg, R., L. M. Dyer, G. E. Garnett, S. T. Pope, and C. Roads. 1989. "Position Papers on Music Representation." In *Proceedings of the International Computer Music Conference.* San Francisco: International Computer Music Association.

Dannenberg, R., P. McAvinney, and D. Rubine. 1986. "Arctic: A Functional Language for Real-Time Systems." *Computer Music Journal* 10(4): 67–78.

Desain, P., and H. Honing. 1986. "LOCO: Composition Microworlds in Logo." In *Proceedings of the International Computer Music Conference.* San Francisco: International Computer Music Association.

Desain, P., and H. Honing. 1988. "LOCO: A Composition Microworld in Logo." *Computer Music Journal* 12(3): 30–42.

Desain, P., and H. Honing. 1992. "Tempo Curves Considered Harmful." In *Contemporary Music Review.*

Desain, P. 1990. "Lisp as a Second Language." *Perspectives of New Music* 28(1).

Hiller, L., A. Leal, and R. A. Baker. 1966. "Revised MUSICOMP Manual." Technical Report 13. Urbana: University of Illinois, School of Music, Experimental Music Studio.

Honing, H. 1991. "Issues in the Representation of Time and Structure in Music." In I. Cross, ed. *Proceedings of the Music and the Cognitive Sciences Conference, Contemporary Music Review.* London: Harwood Press.

Huron, D. 1990. "Design Principles in Computer Based Music Representation." In A. Marsden and A. Pople, eds. *Computer Representations and models in Music.* London: Academic Press.

Jaffe, D. 1985. "Ensemble Timing in Computer Music." *Computer Music Journal* 9(4): 38–48.

Koenig, G. M. 1970. "Project 2. Computer Programme for Calculation of Musical Structure Variants." Electronic Music Reports 3. Utrecht: Institute of Sonology.

Loy, D. G. 1985. "Musicians Make a Standard: The MIDI Phenomenon." *Computer Music Journal* 9(4): Reprinted in C. Roads, ed. 1989. *The Music Machine.* Cambridge: MIT Press, pp. 181–198.

Loy, D. G. 1988. "Composing with Computers: A Survey of Some Compositional Formalisms and Music Programming Languages." In M. V. Mathews and J. R. Pierce, eds. *Current Directions in Computer Music Research.* Cambridge: MIT Press, pp. 291–396.

Mathews, M. V., and F. R. Moore. 1970. "GROOVE: A Program to Compose, Store and Edit Functions of Time." *Communications of the ACM* 13(12).

Mathews, M. V. 1969. *The Technology of Computer Music.* Cambridge: MIT Press.

Pope, S. T. 1989. "Modeling Musical Structures as EventGenerators." In *Proceedings of the International Computer Music Conference.* San Francisco: International Computer Music Association.

Pope, S. T. 1991. "An Introduction to the MODE: The Musical Object Development Environment." In S. T. Pope, ed. *The Well-Tempered Object: Musical Applications of Object-Oriented Software Technology.* Cambridge: MIT Press, pp. 83–106.

Puckette, M. 1988. "The Patcher." In *Proceedings of the International Computer Music Conference.* San Francisco: International Computer Music Association.

Rahn, J. 1990. "The Lisp Kernel: A Portable Software Environment for Composition." *Computer Music Journal* 14(4): 42–58.

Rodet, X., and P. Cointe. 1984. "FORMES: Composition and Scheduling of Processes." *Computer Music Journal* 8(3). Reprinted in C. Roads, ed. 1989. *The Music Machine.* Cambridge: MIT Press. Also reprinted in S. T. Pope, ed. 1991. *The Well-Tempered Object: Musical Applications of Object-Oriented Software Technology.* Cambridge: MIT Press, pp. 64–82.

Schottstaedt, B. 1983. "PLA: A Composer's Idea of a Language." *Computer Music Journal* 7(2). Reprinted in C. Roads, ed. 1989. *The Music Machine.* Cambridge: MIT Press, pp. 285–294.

Smoliar, S. W. 1980. "A Computer Aid for Schenkerian Analysis." *Computer Music Journal* 4(2): 41–59.

Steele, G. 1990 *Common LISP: The Language.* Bedford, Massachusetts: Digital Press.

Appendix

```
;;; TIME FUNCTIONS MICROWORLD
;;; (C)1991, Peter Desain & Henkjan Honing
;;; Part of the COCO composition system
;;; In Common Lisp (uses loop macro)

;;; basic musical objects

(defun note (&key (duration 1) (pitch 60) (amplitude 0.7))
  "Return an event-list-generator of a note"
  #'(lambda (start factor &aux (stretched-dur (* duration factor)))
      (list (list (list :start start
                        :duration stretched-dur
                        :pitch pitch
                        :amplitude amplitude))
            (+ start stretched-dur))))

(defun pause (&key (duration 1))
  "Return an event-list-generator of a pause"
  #'(lambda (start factor &aux (stretched-dur (* duration factor)))
      (list nil (+ start stretched-dur))))

;;; compound musical objects (time structuring)
```

```
(defun s (&rest elements)
  "Return an event-list-generator of a sequential compound object"
  #'(lambda (start factor)
      (loop for element in elements
            as start-time = start then end
            as (events end) = (funcall element start-time factor)
            append events into result
            finally (return (list result end)))))

(defun p (&rest elements)
  "Return an event-list-generator of a parallel compound object"
  #'(lambda (start factor)
      (loop for element in elements
            as (events end) = (funcall element start factor)
            append events into result
            maximize end into end-time
            finally (return (list result end-time)))))

;;; time transformation

(defun stretch (object amount)
  "Return an event-list-generator of a stretched object"
  #'(lambda (start factor) (funcall object start (* amount factor))))

;;; attribute transformation

(define-modify-macro modify-attribute (operator time-fun)
   modify-time-fun)

(defun modify-time-fun (old-time-fun operator new-time-fun)
  "Return a composition of two time-functions given an operator"
  (time-fun-compose operator new-time-fun old-time-fun))

(defun attribute-transform (keyword time-fun operator object)
  "Return an event-list-generator of a transformed musical object"
  #'(lambda (start factor)
      (destructuring-bind (events end) (funcall object start factor)
        (let ((global-time-fun
               (global-to-local time-fun start (- end start)))
              (events-copy (copy-list events)))
          (loop for event in events-copy
                do (modify-attribute (getf event keyword)
                                     operator
                                     global-time-fun)
                finally (return (list events-copy end)))))))

;;; use of time functions over compound musical objects

(defmacro make-local-time-fun (time-fun start duration)
  "Return code for a time-function of unknown start and duration"
  `#'(lambda (local-start local-duration local-progress)
       (funcall (global-to-local ,time-fun ,start ,duration)
                local-start local-duration local-progress)))

(defmacro let-time-funs-over-compound (bindings expression)
  "Establish bindings of time-functions over compound object"
  (reduce #'(lambda (&rest list)
              (cons 'let-time-fun-over-compound list))
          bindings :initial-value expression :from-end t))

(defmacro let-time-fun-over-compound ((var fun) expression)
  "Establish binding of time-function over compound object"
  (let ((start (gensym))(dur (gensym)))
    `(let* (,start ,dur (,var (make-local-time-fun ,fun ,start ,dur)))
       #'(lambda (start factor)
           (destructuring-bind (events end)
                               (funcall ,expression start factor)
             (setf ,start start ,dur (- end start))
             (list events end))))))
;;; time function utilities

(defun time-funcall (time-fun-or-constant start duration progress)
  "Return a constant or apply time-function to its arguments"
  (if (functionp time-fun-or-constant)
    (funcall time-fun-or-constant start duration progress)
    time-fun-or-constant))

(defun time-fun-compose (operator &rest time-funs)
  "Return a time-function that applies operator to results of time-funs"
  #'(lambda (start duration progress)
      (apply operator
             (mapcar #'(lambda (time-fun)
                         (time-funcall time-fun start duration progress))
                     time-funs))))

(defun time-fun-concatenate (time-fun-1 time-fun-2 proportion)
  "Return a concatenation of two time-functions given a proportion"
  #'(lambda (start duration progress)
      (if (<= progress proportion)
        (time-funcall time-fun-1
                      start
                      (* duration proportion)
                      (/ progress proportion))
        (time-funcall time-fun-2
                      (+ start (* duration proportion))
                      (* duration (- 1 proportion))
                      (/ (- progress proportion) (- 1 proportion))))))

(defun global-to-local (time-fun start duration)
  "Return a global time-function that can be referenced locally"
  #'(lambda (local-start local-duration local-progress)
      (let* ((time (+ local-start (* local-duration local-progress)))
             (progress (/ (- time start) duration)))
        (time-funcall time-fun start duration progress))))

;;; sine function used in figure 11

(defun make-sine ()
  "Return sine function with state"
  (let ((phase 0) old-time)
    #'(lambda (time frequency)
        (when old-time
          (incf phase (* 2 pi (- time old-time) frequency)))
        (setf old-time time)
        (sin phase))))

;;; graphical score output

(defun draw (musical-object &key (resolution 0.1))
  "Draw a musical object"
  (loop for note in (first (funcall musical-object 0 1))
        do (apply #'draw-note resolution note)))

(defun draw-note (resolution &key start duration pitch amplitude)
  "Draw a note using the time-function or constant of the attributes"
  (loop
    for count from 0 to (floor (/ duration resolution))
    as time = (+ start (* count resolution))
    as old-pitch = (time-funcall pitch start duration 0)
    then new-pitch
    as old-amplitude = (time-funcall amplitude start duration 0)
    then new-amplitude
    as old-time = start then time
    as progress = (/ (- time start) duration)
    as new-pitch = (time-funcall pitch start duration progress)
    as new-amplitude = (time-funcall amplitude start duration progress)
    do (format t "~%~2,1,5$~2,1,7$~2,1,7$" time new-pitch new-amplitude)
    do (draw-air-brush old-time old-pitch
                       time new-pitch
                       old-amplitude new-amplitude)))

(defun draw-air-brush (x1 y1 x2 y2 shade1 shade2)
  ;; to be provided by the user, adapted to the specific graphical system
  ;; draws a parallelogram with vertical left and righthand sides
  ;; (x1, y1) is mid left, shade1, (x2, y2) is mid right, shade2
  )

;;; examples of use (draws pictures as in figures 4b, 4c and 4d)

(defun oscillator (offset frequency depth)
  #'(lambda (start duration progress)
      (+ offset
         (* depth
            (sin (* 2 pi duration progress frequency))))))

(defun sine-glissando (key depth)
  #'(lambda (start duration progress)
      (+ key
         (* depth (sin (* 2 pi progress))))))

(defun sine-ornament (key count)
  #'(lambda (start duration progress
                  &aux (relative-time (* duration progress)))
      (+ key
         (if (< relative-time count)
           (sin (* 2 pi relative-time))
           0))))

(defun vibrato-example ()
  (s (note :duration 1 :pitch 58)
     (p (s (pause :duration .5)
           (note :duration 2 :pitch (oscillator 61 1 1)))
        (note :duration 2.5 :pitch (oscillator 64 1 .5)))))

(defun glissando-example ()
  (s (note :duration 1 :pitch 58)
     (p (s (pause :duration .5)
           (note :duration 2 :pitch (sine-glissando 61 1)))
        (note :duration 2.5 :pitch (sine-glissando 64 .5)))))

(defun ornament-example ()
  (s (note :duration 1 :pitch 58)
     (p (s (pause :duration .5)
           (note :duration 2 :pitch (sine-ornament 61 1)))
        (note :duration 2.5 :pitch (sine-ornament 64 .5)))))

; figure 4b:
;(draw (s (vibrato-example) (stretch (vibrato-example) 2)))

; figure 4c:
;(draw (s (glissando-example) (stretch (glissando-example) 2)))

; figure 4d:
;(draw (s (ornament-example) (stretch (ornament-example) 2)))
```